Conceptual object-oriented database: A theoretical model

M.L. Hines *

Computer Science Telecommunications Program, University of Missouri–Kansas City,
55100 Rockhill Road, Kansas, MO 64111, USA

Received 1 May 1995; received in revised form 4 January 1996; accepted 20 May 1997

Abstract

The evolution of object-oriented databases (OODBs) has not been accompanied by an equivalent evolution of theoretical models, leaving the foundations for OODBs ill-defined. Design of OODBs has been hampered by the lack of design techniques/tools which correspond to the theoretical model. This paper defines a core conceptual object-oriented database (COODB) model providing a foundation and framework for theoretical research. Structural and behavioral definitions for an object, a class, and a class hierarchy are given. A specification hierarchy is introduced as a design tool, and a messaging component is defined which enables asynchronous and synchronous interaction. © 1998 Elsevier Science Inc. All rights reserved.

Keywords: Object-oriented databases; System specification; Metadata evolution

1. Introduction

A data model is a logical organization of the components (entities) of an enterprise, the constraints on those entities, and the relationships among them. Data models provide a theoretical foundation or context for the representation and manipulation of data. A database is an organized integrated collection of data; most are organized according to one logical data model.

---

* Corresponding author. Tel.: +1 816 235 1193; fax: +1 816 235 5159; e-mail: hines@estp.umkc.edu.
Originally database models—such as network, hierarchical, and relational—were developed when the primary concern of application designers was conventional business data-processing applications, e.g., inventory control, payroll, accounts, etc. [21]. Emerging data representation needs—e.g., CAD, CAE, CASE, CAM, knowledge-based systems, and multimedia systems—have highlighted the inadequacies of these traditional database models. Some of the limitations include not allowing nested construction of complex designs, enforcement of semantic consistency, capture of version semantics, database design, or representation of procedural and declarative knowledge as a unit.

One approach to address these inadequacies has been to use the data model based on object-oriented concepts. Confluence of existing database technology and object-oriented concepts allows the modeling capabilities and enforcement of semantic consistency necessary for emerging data representation needs. There exist a variety of commercial/research object-oriented database (OODB) systems which combine object-oriented concepts and features of traditional database systems in different ways, e.g., Itasca (Orion), GemStone, Iris, O₂ and Encore [12,17,22,36]. In addition, both the National Institute of Science and Technology and the Object Database Management Group have proposed object database standards (ODMG-93 and OIM) to specify common object database interfaces allowing interoperability among the various object databases [9,50]. The focus of these efforts at standardization has been on commonality of interface protocols and terminology [9,50]. While these are necessary activities, they do not provide a theoretical data model that facilitates data modeling research.

In theory, OODBs offer a number of advantages: increased data representation capabilities including representation of abstract data types or meta-data, navigational access to data, encapsulation of procedural and declarative knowledge, management of class extensions, dynamic class definition, etc.

In reality, however, the advantages offered by many of the OODBs fall short of expectations—often because the OODB model is an extension of an object-oriented programming language such as C++, Smalltalk, or Lisp [26]. Because many of the commercial/research OODBs are closely tied to a specific programming language, the database acts very much as a part of the application programming system rather than as an independent data repository [26]. Current database models are not independent data repositories in that they do not store meta-data or procedural knowledge within the database; rather the procedural knowledge (methods) are stored outside the database in regular files [26]. Meta-data are also not treated as part of the database, so the user cannot access class properties. The representation of complex objects is often superficial [26] and limited by the ability of the underlying programming language to represent relationships between objects. Thus, using programming languages as a theoretical data model for an OODB severely constrains data modeling ca-
pabilities and does not allow the needed access to meta-data which is essential for research and use of OODBs.

Little work has been done to provide support for designing OODBs. The need for friendly and efficient design aids for the logical and physical design of OODBs is significantly stronger than that for relational databases [21]. Existing OODB systems do not support design as part of a data model, or supply design tools beyond browsers and visual aids, as part of their environments [12,16,22,36]; again this can be traced to the data model used. Extending programming languages with persistence does not provide a design environment.

A formal conceptual OODB model integrates concepts from traditional database models and object-oriented concepts. Traditional database models contribute a formalization of theory; the object-oriented concepts contribute the capability for relatively rich semantic representation. The preservation of data independence and the level of semantic capture is important for OODBs. Typing capabilities and the data abstraction level of the OODB are important in preserving data independence, raising the level of semantic capture, defining database level data manipulation capabilities, defining finer granularity for concurrency control, and identifying possible security levels. Formalizing an OODB model provides the foundation for incorporating the above concepts in developing OODBs. Researchers have raised questions about the feasibility and possibility of formalizing a complete OODB model [5,22]. The two primary positions regarding these questions appear different on the surface; upon examination, they both support the development of an object-oriented data model, which includes certain object-oriented concepts and leaves the others as extensions [5,22].

To date there is not a common theoretical model – or core data model – for the components of OODB systems, the OODB system itself, or their inheritance strategies [9,19,22,26,50]. This deficiency hinders both research and adoption of OODBs by industry [26]. A core model, based on a mathematical foundation, is necessary to support development and implementation of OODBs, and to provide a foundation for theoretical investigations [5,23]. In addition, this core model should facilitate database design as a database function, i.e., design must be integrated with the database itself rather than an entirely separate consideration.

This paper defines a core conceptual object-oriented database (COODB) model. The term ‘conceptual’ is used to indicate that the model does not address implementation issues. Rather it provides a common, minimal, theoretical framework within which OODBs can be benchmarked with respect to representational issues and meta-data support. One of the advantages of defining an object-oriented data model in an object-oriented fashion is that the advantages of the object-oriented concepts can be incorporated. Specifically, by providing “basic sockets”, a plug and play approach can be used for facets
which are domain dependent, e.g., class change propagation techniques, distributed issues, security issues, etc. The COODB model—based on set theory—defines the structure of core components including object, class, and class hierarchy; structural constraints; and the components' behaviors including message passing. This work includes and, in some cases, extends the generally understood usage of classes as templates for instantiation of objects, the general notions of methods, instance and class variables, a class hierarchy, and the use of message passing [9,42].

A unique and significant contribution of this work is the definition of a specification hierarchy, as part of the core COODB model, to allow flexibility during design, and stability during use of the database. The specification hierarchy as a design tool, allows careful a priori characterization of objects and classes, and determination of the validity of classes and relationships during development. It allows specification of three types of relationships with respect to inheritance: inheritance, e.g., class A is a subclass of class B; prohibited relationships, e.g., class A is prohibited from being a subclass of class B; and future relationships, e.g., class A and class B have some relationship which may be specified in the future—at present that relationship is only known to exist and is not known in detail. The specification hierarchy provides a design platform which is flexible yet preserves the integrity of the user’s view through explicit connection with the class hierarchy.

The structure of the paper is as follows. Section 2 reviews the current OODB systems, especially the formality of their foundations. Section 3 presents the COODB structures and their constraints; Section 4 presents the COODB behaviors building up to the COODB model components. Sections 5 and 6 conclude the paper with a summary of the contributions of the work and an overview of open problems and future work in the COODB model. To keep the presentation easy to follow, the formal definitions have been removed from the text and compiled in Appendix A.

2. Background

Many products claim to be OODBs. There is little consistency in terminology from one product to the next, and there are no formal models to support these products [1,10,22,23,29,34,37,47-49]. Many of these systems consist of a “layer” of software interfacing between the user and a traditional relational database system [30,36]. Currently, there are many OODB systems either in development or commercially available, including Vbase [2,3], VISION [7,8], Cactis [18] (many of the ideas and concepts of Vision are incorporated into Cactis), Trellis/Owl [35], POSTGRES [38,39,43-46], Encore [17], GemStone [30,31,36], Iris [12], ORION/Itasca [4,23-25], and O2 [27,28]. Three of these systems have been chosen, as representative of the breadth of OODB
implementation alternatives, for further discussion: POSTGRES, Itasca, and O2.

POSTGRES is an extensible database management system developed at the University of California [38,39,43–46]. The POSTGRES data model is an extended relational model supporting general mechanisms for abstract data types, data of type procedure, and rules [38]. These mechanisms are used to support complex objects or to implement a shared object hierarchy for an object-oriented programming language [39]. They are also used to simulate a wide variety of semantic and object-oriented data modeling constructs including aggregation and generalization, and attributes that reference tuples in other relations. Inheritance is simulated using an inherits-clause to share the definition of attributes and procedures. POSTGRES supports a query language, POSTQUEL, which is a generalized, extended version of QUEL. Declarative and procedural knowledge can be captured together, encapsulation is not supported and knowledge independence of the tuple structure of the data is compromised.

The developers of POSTGRES explicitly state the preference of not referring to POSTGRES as object-oriented [38]; it is included here as an example of a database system which attempts to capture the rich semantic capabilities of the object-oriented concepts by extending the relational model.

ORION/Itasca is an OODB system developed at MCC in Austin, Texas [4,21,25]. It was originally developed as a research vehicle for development of database technology for object-oriented applications in CAD/CAM, artificial intelligence, and office information systems. ORION/Itasca was developed in Common LISP on the Symbolics LISP machines and has also been ported to the SUN3 workstation using the UNIX operating system. The application interface to ORION is an object-oriented extension to LISP and the ORION/Itasca system itself extends Common LISP with object-oriented programming and database capabilities [25]. ORION/Itasca views both classes and instances of classes as objects to insure uniformity in message handling. It also supports two types of variables: shared-value variables and default-value variables. Shared-value variables are similar to class variables. Default-value variables take a specified default value if a value is not specified by the object. ORION/Itasca supports class hierarchies and a form of multiple inheritance where an object can belong to only one class; if an object needs to belong to more than one class, a new class is created which has as its superclasses all those needed for inheritance by that object. ORION/Itasca has no formal model or design facilities.

The Altair group in France designed an OODB system called O2 which is a type system defined within the framework of a set-and-tuple data model [27,28]. Target applications for this system are traditional business applications, transactional applications (excluding very high performance applications), and office applications. The O2 system allows recursively defined
types. The O₂ system supports inheritance by allowing objects of different structures to share methods, and by supporting an inheritance hierarchy which allows sharing of common structure.

Set and tuple constructors are used to define complex objects, which are grouped into types. Types define a minimal common structure and common behavior, and are an abstraction encapsulating data and operations in the same structure. The static component of a type is the type-structure, which classifies objects with respect to their structure. Operations are referred to as methods. Interpretations are functions associating subsets of a consistent set of objects to type structure names. The O₂ system has been implemented on a Sun workstation running the Unix operating system. The O₂ system provides some formality in its definition, but fails to address fundamental questions regarding inheritance in cyclic graphs.

Most of the extant OODB systems have some form of concurrency control and type extensibility, and some notion of an inheritance hierarchy. There is little formal theory and no theoretical conceptual model support for any of the systems [1,10,22,23,29,34,37,47,49]. The power of object-oriented concepts is being tapped by these products but realization of the full potential of applying object-oriented concepts to the field of database should be done by first providing the formal theory and modeling concepts of an OODB before launching into a vast array of products [37]. Table 1 summarizes the three OODB systems discussed.

### 3. COODB model

The COODB model components include: (1) objects which are instances of classes and represent or model real-world entities residing within the domain of interest (as opposed to those addressing implementation issues); (2) classes which are elements or nodes of the class hierarchy and serve a template role in definition of objects; (3) a class hierarchy which provides the structure of

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Comparison of OODB systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object representation</td>
<td>POSTGRES</td>
</tr>
<tr>
<td>Communication</td>
<td>tuple const.</td>
</tr>
<tr>
<td>Methods</td>
<td>function call</td>
</tr>
<tr>
<td>Language</td>
<td>POST-QUEL</td>
</tr>
<tr>
<td>Inheritance</td>
<td>form of multiple</td>
</tr>
<tr>
<td>Available types</td>
<td>user def.</td>
</tr>
<tr>
<td>Implemented type</td>
<td>proto-type</td>
</tr>
<tr>
<td>Formal model</td>
<td>yes</td>
</tr>
<tr>
<td>Level of data manipulation</td>
<td>outside db</td>
</tr>
</tbody>
</table>
a COODB database; and (4) a specification hierarchy which is used in designing the COODB structure and the paths of inheritance. Structural constraints define inter-relationships and intra-relationships among the COODB components and subcomponents.

3.1. Primitive domains and basic structures

In this section primitive domains (and sets which are basic structures) are defined as denumerable sets for which the operations of equal and unequal hold. With the basic structures, the primitive domains provide the basis for the structural definitions of the COODB components.

All keys in a COODB — object, class, variable, message, and method — are unique and unchangeable. Keys may not be reused even if the object, class, variable, message, or method they originally represented has been deleted. These keys are drawn from *identifier* domains which are sets of atomic elements. The identifier domains are: *obj_id_dom* — object identifiers; *class_id_dom* — class identifiers; *var_id_dom* — variable (class and instance) identifiers; *method_id_dom* — method identifiers; and *msg_id_dom* — message identifiers. There is one structural constraint defined on the primitive domains. The identifier domains must be disjoint to prevent confusion and the necessity of dynamic run-time differentiation (Fig. 1).

The *var_struct* is a basic structure used to define variables for objects and classes; it is a three tuple (vkey, type, value) where vkey ∈ var_id_dom; type ∈ class_id_dom; value ∈ obj_id_dom (Fig. 2(a)). All variables are of type *var_struct*. A variable set, *var_set*, is a set of *var_struct*.

A *method* represents the procedural knowledge of a class or an object. A method is composed of a *signature* containing a method key, which uniquely identifies the method, and *parameters* which represent the formal parameters of the method; and a *body*, composed of functions defining the operations or actions the method invokes and executes (Fig. 2(b)). The currently defined kinds of functions for a method include state changing functions, calling functions (which generate a message), and functions which return a value. These kinds of functions were chosen as a semantically minimal set of behaviors. Other kinds of functions can, of course, be added as dictated by the domain and within the theoretical framework.

Inheritance in a COODB is the process of acquiring methods and variables, by a class from another class, or by an object from a class. An *inheritance relationship*, \( R(x,y) \), exists between an object, \( x \), and a class, \( y \), or a class, \( x \), and another class, \( y \), such that if \( R(x,y) \) is an inheritance relationship, then \( x \) inherits from \( y \). A *superclass* in an inheritance relationship, \( R(x,y) \), is the class \( y \). The *class_pair* structure, which is used in defining relationships in the class and specification hierarchies, is an ordered pair composed of two class identifiers.
Fig. 1. Identifier domains.

Fig. 2. (a) var_struct (b) method.
3.2. Message passing

Message passing is the means by which objects and classes communicate with one another in the COODB. An object or class sends a message to another object/class or to itself requesting the application of a method. Generally, only objects are thought of as receiving messages; in the COODB model, classes also may receive messages, e.g., to create or instantiate an object. In addition, a response to a message may need to be forwarded to yet a third object/class, rather than to the message sender, e.g., as in the case of process flow or control. Various system architectures should be accounted for in the messaging design, e.g., local, distributed, client-server, etc. The messaging design should be as broad as possible when considering how messages are controlled or “typed”. A message (Fig. 3) is a seven-tuple, \((\text{msgkey}, \text{sender}, \text{receiver}, \text{recipient}, \text{contents}, \text{control}, \text{response_to})\). It is uniquely identified by its system-assigned message key \((\text{msgkey})\).

Senders and receivers of messages are either classes or objects. The recipient of a message identifies the object/class which is to receive the response to the message (if there is one) (Fig. 4). This allows the response to be sent to yet a third object/class. The contents of the message either identify a method to be
Fig. 4. Senders, receivers, and recipients of messages.
executed, or contain a value, which may be a response from the execution of a method. The control of a message identifies whether the message requires an asynchronous (asynch) or synchronous (synch) response or contains a response to a previous message (response). If it is a response message, then the response_to field contains the message identifier of the original message. This allows matching of messages and responses via message identifiers.

Objects/classes may need to interact with other objects/classes in a synchronous or asynchronous manner. The structure of messages facilitates both synchronous and asynchronous interaction. If an object/class, say A, sends a message for which it must have a reply before it can proceed, then the recipient field would represent A and the control field of the message would be synch (Fig. 5(a)). If an object/class needs to interact asynchronously, the control field of the message would be asynch (Fig. 5(b)). If the message is a response to an earlier message, the control field of the message would be response and the response_to field would contain the identifier of the message to which the current message was a response (Fig. 5(c)). The contents field specifies the method requested by the message or the value being returned by the message. The ability to interact synchronously or asynchronously is one factor which will facilitate distributability of a COODB.

As messages are sent and received there must be a repository for messages which are not immediately responded to, e.g., the object or class is already busy. A message queue identifies an object or class receiving a message and contains the queue of messages for that object or class (Fig. 6). Message queues are part of the overall environment within which objects and classes reside, i.e., the class hierarchy.

3.3. Object structure

In the COODB model, entities are characterized by object, the basic object-oriented component. Objects are unique entities in a COODB (Fig. 7), with their own identity and existence, and are represented by a five-tuple, (OKey, OState, OPredecessor, OParent, OMethod). Objects are identified by a unique system-assigned key, OKey. The use of a system key allows objects to be referred to regardless of their attribute values, supports referential integrity [11], allows navigational access, and provides an advantage over record-oriented data models in which objects, represented as records, can be referred to only in terms of their attribute values [20].

OState represents the state of an object, o, as instance variables inherited from o's parent class, and the types and values of those inherited instance variables. The state of an object may be changed only by application of a method to an instance variable.

A version of an object can represent either another object whose state was originally (and deliberately, as opposed to coincidentally) the same as the first
Fig. 5. (a) Synchronous message (b) Asynchronous message (c) Response message.
object, or the temporal representation of a single object where multiple states of the object are represented. Several OODB implementations represent versioning by allowing objects to instantiate other objects, e.g., [7,8,40]. Since classes are distinguished from objects semantically as templates, versioning via instantiation by objects is not allowed in the COODB model. Objects are classified as versionable or non-versionable via their class membership. If an object is non-versionable, no historical or temporal information is kept - only that of the current state.

The \( \text{OPredecessor} \) of an object, \( o \), represents the unique object which temporally preceded \( o \) in creation, and of which \( o \) is a version – i.e., \( \text{OPredecessor} \) contains the \( \text{OKey} \) of the object from which \( o \) drew its initial state (the \( \text{OState} \) when \( o \) was instantiated). \( \text{OPredecessor} \) allows representation of versions of objects within a COODB. There are several versioning options which an implementation could choose: an object may be a version of a single predecessor (tree structure), a member of a sequence of versions (has a single predecessor

\[
\begin{align*}
\text{OKey} & : \text{obj\_id\_dom} \\
\text{OSState} & : \text{var\_set} \\
\text{OPredecessor} & : \text{obj\_id\_dom} \\
\text{OParent} & : \text{class\_id\_dom} \\
\text{OMethod} & : \{\text{method\_id\_dom}\}
\end{align*}
\]

Fig. 7. Object structure.
and is a single successor), or its OPredecessor set may be null, either because it is the first object in the version or because versions are not kept for objects of the class of which o is an instance. It is also possible to extend the definition of OPredecessor to be a set of objects from which an object, o, draws its initial state but first the problem of reconciliation of multiple object states would have to be resolved. Referential integrity can be preserved through the use of object versioning given that the COODB starts a versioning transaction from a referentially integral state, i.e., use of non-attribute keys which are unchangeable and unique preserves referential integrity since versioning an object does not necessarily imply that references to the initial object are automatically updated to the new version. The propagation of object updates remains as an open problem which is dependent upon the database domain.

Methods are functions whose application can cause the change of an object's state or the performance of a specified behavior, e.g., sending a message or returning a value. An object contains a set of method identifiers, OMethod, which contains the keys of methods recognized by the object.

Objects are unique instantiations of a single class. The OParent field identifies the class of which a given object is an instance (parent class). The method set, OMethod, and instance variables, OState, of an object are inherited from the parent class represented by OParent.

3.4. Class structure

Conceptually, a class represents a collection of user-world entities which are similar within the user’s modeling perception and, in a sense, can be used to manage that collection. Structurally, a class is a passive template representing a group of objects having certain properties in common; this role is the generally accepted role of a class in an object-oriented system. In the COODB model, a class also is active, since a class may receive messages, e.g., requests to instantiate an object, add an instance variable, add a method, etc. A COODB class also represents aggregate data – through its class variables – about its instances, and can be queried with respect to that data. A COODB class is represented by an eight-tuple, (CKey, Instance_Ids, Instance_Var, Class_Var, Method_Set, Superclass_Set, Instances, Versions) (Fig. 8).

Classes are uniquely identified by their class key, CKey. Unlike object keys, however, class keys are not necessarily system assigned but rather can be assigned by the designer within a uniqueness requirement – i.e., all CKeys must be unique.

Instance_Var and Class_Var represent the set of instance variables and the set of class variables, respectively. Class_Var and Instance_Var are initialized to empty, and are filled either by definition of superclasses or by definition of variables. The value of any variable must be of the specified type, i.e., from the extension of the class identified as the variable type.
Semantically, class variables and instance variables represent declarative knowledge about a group of entities, e.g., generalized or aggregated declarative knowledge. Representation of the commonality of a group of entities has been addressed in many different ways [6] and includes decisions about how to model or represent exceptions to generalizations or aggregations, e.g., elephants which are not gray in color. There has been no universally accepted definition of class variables and instance variables, resulting in a wide diversity of use [32], particularly with respect to representation of exceptions to generalizations or aggregations of knowledge.

Several alternatives of representation were examined before defining the representation of exceptions to declarative knowledge of a class for the COODB model, and subsequently the semantic definitions of class variables and instance variables [15]. Instance variables represent properties of individual instances or objects of the class, while class variables represent aggregation properties about subsets of instances or objects of the class, e.g., the number of green-eyed elephants, average salary, etc. Values of instance variables can be changed through message passing. Values of class variables are indirectly changed to reflect changing demographics of the class, possibly through probes and daemons on methods whose execution affects the class variable [13]. Values
of class variables are initialized to null. The null value for variables, either instance or class, represents "value unknown" or "value not applicable".

Method_Set contains the methods a class and/or its instances may invoke and execute. The set of methods, Method_Set, is filled either through definition of the Superclass_Set or by explicit definition of methods. Method_Set cannot be left empty, since a class without methods is a completely static class incapable of any behavior, including creation of objects.

Methods and class/instance variables may be designed into a class or may be inherited from a superclass. The Superclass_Set, of a class, c, contains the CKeys of the classes of which c is a subclass. That is, all of the classes represented in c.Superclass_Set are superclasses of c, and c inherits properties from them. This allows multiple inheritance within the COODB but does not directly address issues such as name clashes, parent precedence, repeated inheritance, or the various anomalies of multiple inheritance. Name clashes and parent precedence are implementation issues which must be addressed within the domain of application. These two issues in general arise because of the limitations of the programming languages and/or environments within which an object-oriented system is implemented. The COODB does not dictate resolution mechanisms but rather allows the semantics of the domain of application to dictate their resolution within the framework of the COODB model. Repeated inheritance is addressed by the structural constraints which are presented in the next section. The various anomalies of multiple inheritance, e.g., ambiguity, inconsistency, instability, are addressed by an evolution mechanism [16].

Often classes in object-oriented systems are designated as abstract if they are "place holder" classes, i.e., classes used to build the inheritance or type hierarchy [23]. Classes in the COODB model are explicitly defined as being instantiable or uninstantiable (abstract) through the Instances field. Instantiable means that a class may have associated with it a set of objects which is not necessarily empty. If a class is uninstantiable, its set of objects must be empty. Instance_Ids contains the OKeys of the objects which are instances of the class, or the extension of the class. If the Instances field of a class, c, is false, then the Instance_Ids set must be empty.

Classes in the COODB model are explicitly designated as being versionable or non-versionable through the Versions field. If a class c is versionable, the domain of OPredecessor of the objects of the class is a subset of c.Instances, or for an instance of a non-versionable class, o.OPredecessor = null.

3.5. Class hierarchy

The class hierarchy defines the structure of a COODB, contains a set of classes and a set of objects, and is represented by a four-tuple, (O, C, Message_Q, Response_List) (Fig. 9). A graphical representation of the class hierarchy can be obtained where the classes represent the nodes of the graph and
the Superclass_Sets of the classes can be used to derive the edges of the graph. All class hierarchies contain at least a root class, which has no superclasses. A class hierarchy can have only one root class. The specification of the root class is obviously dependent upon the domain of application but should contain at least those methods and structures necessary for the operation of the system, e.g., system structures and meta-behavior. The class hierarchy also contains structures for message passing, including a message queue and a response list.

The classbase, C, contains the classes available to users through the class hierarchy, and is initialized as containing root_class. Classes are related to one another through inheritance relationships represented by the presence of the CKey of a class in another class's Superclass_Set. These relationships can be derived by the function Edges.

The set returned by the function Edges is composed of ordered pairs of CKeys representing inheritance relationships and defining the class hierarchy as a directed graph. In the COODB class hierarchy an ordered pair in the set returned by the function Edges has the following meaning: given an ordered pair, (class1, class2), class1 is a subclass of class2, and inheritance occurs from class2 to class1 but not vice versa. Connectedness and the acyclicity constraints restrict the structure of the class hierarchy to a connected graph, and restricts a class from being a superclass of itself.

Several constraints provide closure to the class hierarchy and prevent a COODB system from containing anomalous references to non-existing entities. First, all members of the Superclass_Set of a class, c, must also be members of the classbase, C, of the class hierarchy, H. Second, the type of any instance/class variable, x, of a class, c, of the class hierarchy, H, must be a class represented in the classbase, C, of H.
The objectbase, \( O \), is a set of objects, and contains all objects in the class hierarchy; all objects must have unique keys.

The Instance_Ids membership constraint and the unique object parent constraint partition the objectbase, \( O \), such that each object is an instance of a unique parent, i.e., an object is an instantiation of a single class.

The composition of the set OMethod of an object is constrained to only keys of methods contained in the Method_Set of the object's parent class. This constraint prevents addition of methods to an object, where the added method(s) is not part of the parent class, thus preserving the integrity of inheritance.

The instance variable set of an object is constrained to contain only instance variables defined in the object's parent class. This prevents addition of instance variables which are not part of the parent class, and which thus may not be part of a precisely defined COODB system.

The predecessor set, OPredicate, of an object is constrained to the set of Instance_Ids of the parent class. An object cannot draw an initial state from another object which is not an instance of the same class since this would imply different state variables.

The Message_Q is a set of message queue structures, msg_queue, such that received messages are stored by CKey/Okey of the receiver (Fig. 10). This allows messages to be sent and received even if the receiver of the message is busy. Messages can only be sent to or received from classes or objects which are part of the COODB system; all messages have unique identifiers, and all

![Fig. 10. Message_Q structure.](image-url)
classes and objects which are part of the COODB system are represented in the message queue structure, *Message Q*.

The *Response List* is a list of messages, ordered by message key, which are replies to previously sent messages (Fig. 11). Classes and objects check the response list for responses to messages they have sent.

Attaching the messaging structures and the objectbase to the class hierarchy structure allows a simple multicontext or distributed messaging model to be built into the COODB. It also allows simpler or subset messaging models to be built and conversely much more complex models to be built. The ability to "plug and play" different messaging models based upon the COODB framework allows a variety of message queuing schemes to be satisfied without restricting all systems to a single scheme.

### 3.6. Specification hierarchy

Since it is available to users, the class hierarchy must be relatively static in nature particularly with respect to evolution of classes. However, this does not allow a COODB to be as flexible and dynamic as may be needed. Flexibility is primarily needed during design of a COODB, which may be an ongoing process throughout the life of the COODB. With the objective of increased flexibility for designers while also supplying relative stability for users, the *specification hierarchy* is introduced and defined as a design tool for the COODB model (Fig. 12). The specification hierarchy contains a class hierarchy and other classes and relationships not found in the class hierarchy component. The set *Nodes* in the specification hierarchy includes classes available to users and classes still being designed and not yet available to the user. The set *Edges* in the specification hierarchy includes inheritance relationships available to users and inheritance relationships still being designed and not yet available to the user (Fig. 13). Once design of a specification hierarchy *Node* or *Edge* is completed, it can be added to the class hierarchy for user availability. Every class and inheritance relationship found in the class hierarchy is also found in and originated in the specification hierarchy.

Each specification hierarchy contains a class hierarchy, *H*, as defined in the previous section. There is only one class hierarchy in any given specification hierarchy; it is perfectly feasible for the class hierarchy to be partitioned into subsets should a need arise, e.g., for security purposes.

A class contained in the set *Nodes* may be a class which is available to the user – and part of the class hierarchy – or may be a class still being designed.
Fig. 12. Specification hierarchy structure.

Fig. 13. Conceptualization of specification hierarchy.
Since the classbase of a class hierarchy is a subset of the set Nodes of the specification hierarchy and all classes represented by the set Nodes of the specification hierarchy have a unique key, so do classes represented by the classbase of the class hierarchy.

Relationships in the set Edges are relationships which can be directly incorporated into the class hierarchy by adding the component classes to the classbase of the class hierarchy (if they are not already there) and adding the superclass of the Edges relationship to the Superclass_Set of the class of the Edges relationship. For example, if the class-superclass relationship (C1, C2) is an element of Edges and that relationship is to be made available to users by moving it to the class hierarchy, then class C1 and C2 would be added to the class hierarchy’s classbase (if they were not already there), and the CKey of class C2 would be added to the Superclass_Set of class C1 (Fig. 14).

The set of Prohibited relationships defines those inheritance relationships not allowed to exist in the class hierarchy, H. The ability to prohibit certain relationships from existing can increase the semantic capture and integrity of relationships within a COODB, i.e., there are real-world relationships which are, for various reasons, prohibited.

The Future set contains relationships recognized to exist between real-world entities and enterprises but not currently part of the COODB. By allowing representation of possible future relationships between classes in the class hierarchy, H, and/or classes in the specification hierarchy, S, planning for incremental growth of the COODB can be facilitated in a more organized manner; relationships between current classes and potential classes can be recognized at the point when the current class is defined. While not all unknown relationships in the set Future may be represented, use of the Future set provides a tool for initial definition of relationships. The caveat to using the Future set is to recognize the possibility, and probability, of its incompleteness.

The set Future is constrained to be symmetrical, since the relationship represented is semantically unknown. The sets Edges, Prohibited, and Future are mutually disjoint to prevent overlap of semantically incompatible information (Fig. 13).

The structural definitions and constraints of the COODB components provide declarative knowledge of the COODB model. Procedural knowledge of the COODB model is given by defining the behavior of the COODB components. The preservation of consistency, unambiguity and stability of a COODB is addressed by an evolution system [16].

4. COODB behaviors

Typically, data models focus only on structure; this follows the traditional paradigm of splitting declarative knowledge from procedural knowledge and
Fig. 14. Moving an inheritance relationship from the specification hierarchy to the class hierarchy.
representing the procedural knowledge by an application program or within
the database management system. Shifting from the traditional paradigm to
the object-oriented approach to data modeling requires the representation of
declarative and procedural knowledge as a single unit. This shift also means
that many of the activities or behaviors – whose implementation and mainte-
nance were once the responsibility of the application programmer or the data-
base administrator – now become part of the database itself, leading to an
initial thrust towards active or intelligent databases. Databases are no longer
merely data repositories but now become knowledge repositories. This section
defines the COODB model behavior at a meta-level with respect to a com-
ponent and the behavior all components of that kind exhibit. Behavior in the
COODB model (representing procedural knowledge) is defined by providing
functions for the COODB components; these behaviors must preserve and en-
force the structural constraints.

Objects, as a component, do not have any defined behavior at the meta-lev-
el; rather their behavior is individually derived from their parent class.

Class behavior includes functions for adding/deleting/updating a class vari-
able, an instance variable, or a method. However, these functions are not de-
defined in this work as there are many legitimate interpretations of the
semantics of class property updating. Legitimate interpretations include: all
subclasses receive the change; all instances receive the change; only new sub-
classes created after the change occurred receive the change; only new instan-
ties created after the change occurred receive the change; and a new class is created
(class versioning) [14]. Because of the COODB definition paradigm – i.e., ob-
ject-oriented definition – these class behavior functions fall into the “plug
and play” category. The functions themselves are relatively straight-forward
to include as expansions of the COODB model or as behaviors of a database
management system, once the semantics are selected.

Class behavior also includes a function object-map which maps from an ob-
ject identifier to the corresponding object.

Behavior of the class hierarchy includes object-creation behavior; only class-
ies in the class hierarchy may create objects. Object-creation behavior is defined
by the function instantiate. The instantiate function takes as arguments a class
identifier (CKey of the parent class for the object to be created), and an object
identifier if the class is versionable and the new object is to be a version of an
existing object. If the parent class is not an instantiable class, no object is cre-
ated. The application of the instantiate function to a versionable class returns
an object whose OPredecessor is the identifier of some sibling instance or ob-
ject. The application of the instantiate function to a non-versionable class re-
turns an object whose OPredecessor is null. Creation of an object must
satisfy all the structural constraints, e.g., unique object identity, etc.

The OPredecessor sets of the objects in the objectbase form a directed, but
not necessarily connected, graph. Each object is implicitly related only to itself
(non-versionable parent class with a null OPredecessor set) or to another object (versionable parent class) through its OPredecessor set. Objects in the OPredecessor set are also related to other objects, and so forth. The objectbase can be shown to be acyclic with respect to the graphs formed by the objects and their OPredecessor sets.

The partial ordering on the objectbase can be interpreted as a temporal ordering where the order of instantiation forms a rooted sub-graph with the initial instance as the root, or for non-versionable classes, the only instance. In particular, for partial orders reflective of changes in OState, the partial ordering of the objectbase establishes a relative temporal ordering for participating objects.

Class hierarchy behavior also includes message passing behavior to allow communication between components. Message passing is differentiated semantically from message acceptance in the COODB model. Message passing is the request for application of a specified method, sent by an object (or class) to an object (or class). Method acceptance is the receipt of the message and the subsequent application of the requested method by the receiver. Method application is the invocation and execution of the requested method. Messages may not always be accepted—e.g., the sender of the message may not be allowed to send messages to the intended receiver for security reasons, etc. The protocol for message acceptance is another area which is domain dependent; a designer must choose and then "plug and play" the protocols for security, fairness, etc., including the choice of a single protocol or several protocols. Differentiation between message passing and message acceptance delineates messaging in the COODB model from procedure calling in traditional programming languages. Accepted messages are stored in a message queue until they are retrieved and executed by the appropriate class or object.

Responses to messages are stored in a Response_List until they are retrieved by the recipient of the message. Messages are stored by the message identifier on the response_to field. The function response checks for and retrieves a response from the Response_List by the message key.

The function accept defines acceptance of a message and its subsequent enqueuing in the Message_Q. Acceptance of message as it is currently defined (Appendix A) requires that the message be a valid message, i.e., the message receiver "knows" about the method being requested. This definition could easily be expanded to include other acceptance criteria as required by the application domain, e.g., security constraints. The subsequent application of the requested method is done through functions apply and apply_aux. The function apply retrieves a message from the Message_Q and employs the apply_aux function to recursively execute the functions in the body of the requested message.

The function pass creates and passes a message. Definition of the function pass allows replies to messages to be sent to any object or class within the class.
hierarchy, H. The first two parameters of pass represent the sender and receiver of the message; the third parameter of pass represents the recipient of any reply to the message. Messages must include the specific object or class to which the message is sent. This restriction eliminates the problem of messages being sent to non-existent objects or classes, and also restricts the sending of messages to single entities, either a class or an object. Definition of the function pass allows all semantic interpretations of message passing which may be desired, e.g., object to object, object to class, class to object, and class to class.

The function pass returns a three tuple consisting of a Boolean value indicating if the created message was accepted; a value which may contain the response to the message if the message is synchronous and null otherwise; and a response identifier which returns the identifier of the created message for later use in checking for a response to the message.

The behavior of the specification hierarchy consists of functions which allow addition/deletion/update of: a class; an Edge set member; a Prohibited set member; an Future set member; and addition/deletion/update of a class/edge to the class hierarchy of the specification hierarchy. The class update functions (add-variable, add-method) are not defined for the COODB model, as their semantics are implementation dependent or dependent on the database management system of the COODB. Behaviors defined include functions for property inheritance between superclass and subclass and between a class and its instances. The function inherit_properties causes the inheritance of methods and variables from a superclass to a subclass. The function class_to_object_inherit causes the inheritance of methods and variables from a parent class to its instances or extension.

The next group of functions allow the creation of classes and relationships including inheritance relationships, Prohibited relationships, and Future relationships. Creation and development of classes and relationships occur in the specification hierarchy. Once the development of a class and/or relationship reaches a point where it is deemed "safe" for user access, the class/relationship can be transferred to the class hierarchy and made accessible to users (Fig. 14). The functions to accomplish this transfer are defined; however, to preserve the semantic integrity of the COODB, these functions must operate within an evolutionary environment which guarantees a stable, consistent, and unambiguous system [16]. The creation functions include: create_class; create_edge; create_prohibited; and create_future. The transfer functions include: move_class which makes a class from the specification hierarchy accessible in the class hierarchy; and move_inheritance_relation which makes an inheritance relationship from the specification hierarchy accessible in the class hierarchy.

COODB objects are structural at the meta-level, since there is no meta-behavior specified for objects by the COODB model. All object behavior is inherited by an individual object from its parent class. The final definition of the other COODB components – class, class hierarchy and specification hierarchy
COODB Specification Hierarchy: (spec_hierarchy_struct, inherit_properties, class_to_object_inherit, create_class, create_edge, create_prohibited, create_future, move_class, move_inheritance_relation)

COODB Class Hierarchy: (class_hierarchy_struct, instantiate, response, accept, apply, apply_aux, pass)
  * does not show housekeeping functions for management of Message_Q and Response_List

COODB Class: (cls_struct, object_map)

COODB Object: (obj_struct)

Fig. 15. COODB components.
combines the declarative knowledge or structure of the component with the appropriate procedural knowledge or behavior of the component (Fig. 15).

5. Conclusions

This work has described a core conceptual OODB model including COODB structural components – object, class, class hierarchy, specification hierarchy – structural constraints, and behavior. While the concepts defined include those generally considered to be central to object-oriented systems – object, class, class hierarchy, inheritance, methods, and variables – other concepts are introduced directed at answering needs within the database community, specifically the introduction of the specification hierarchy as a design tool. A foundation for incorporation of procedural knowledge within the database itself is provided by including meta-behavior in the COODB model. Combining procedural knowledge and declarative knowledge allows behaviors previously residing within application programs or the database management system to migrate to the database itself. This removes a level of administration from continual human interaction, and allows focus on required functionality as opposed to "housekeeping" activities.

The inclusion of a messaging component helps differentiate message passing from procedure calling and provides a foundation for distributability and parallelism. Both the messaging component and the combination of procedural and declarative knowledge provide background for moving transaction management activities into the database.

In addition, a semantic differentiation between class and instance variables is made which differs from the common default value specification. The incremental definition of the COODB allows relative comparison of COODB implementations, i.e., an implementation may have only implemented the COODB structural definitions while another implementation also implemented the structural constraints, etc.

While there are many OODBs currently on the market, and OIM and ODMG-93 attempt to address implementation and portability issues through establishment of standards [9,50], implementations and standards do not provide a foundation for theoretical research. The formal definition of an OODB model, specifically the COODB, provides a foundation for future research into the underlying theories of the object-oriented concepts in general and OODBs in particular. Part of the future research problems that must be addressed include the controlled evolution of an OODB and how to organize and manage a potentially huge number of classes, objects, relationships, and behaviors. By defining a core data model, these evolutionary and management issues have a basis upon which to proceed. Although this work represents a potentially significant step for OODBs, it is far from being
comprehensive or complete with respect to an OODB system specification. However, because it is specified in an object-oriented fashion, expansion to include other features is possible.

6. Open problems

There are a wide range of potential expansions/future work. The introduction of the specification hierarchy with the prohibited and future relationships necessitated the formal definition of inheritance as an evolutionary process to ensure semantic consistency; this work has been completed [16]. Message passing is currently somewhat limited; expansion of message passing to include the capability to broadcast messages to multiple entities would enhance this feature. In addition, definition of various types of system architectures and their influences/impact on the COODB model and specifically the message passing facilities of the COODB model is needed. Much of the power of the message passing facilities can only be elucidated within the specification of a particular system architecture, e.g., distributed, client-server, etc. The messaging model also needs to be expanded to partition access to meta-level behavior. Many object-oriented systems include a type system or hierarchy in addition to the class hierarchy. While a type system has not been initially defined for the COODB model, it is another possible expansion area. Definition of composite objects requires closely examining security and concurrency with respect to object/class views. Other areas of on-going or future research include: use of recursive data types; security model development; concurrency control; user interfaces and query languages; class versioning; attachment of rules; coalescence of object versions; and implementation.

Acknowledgements

We wish to acknowledge the contributions of several individuals in the preparation of this paper. Dr. Elizabeth Unger of Kansas State University was instrumental in the development of the original work. Dr. Vijay Kumar of the University of Missouri–Kansas City contributed enormously in the writing process, particularly with respect to clarification of several of the more detailed concepts. The comments of the anonymous reviewers were extremely helpful and encouraging.
Appendix A

Defn 1. \texttt{obj\_id\_dom} identifiers for objects such that \texttt{obj\_id\_dom = \{identifier\}}.

Defn 2. \texttt{class\_id\_dom} identifiers for classes such that \texttt{class\_id\_dom = \{identifier\}}.

Defn 3. \texttt{var\_id\_dom} variable identifiers such that \texttt{var\_id\_dom = \{identifier\}}.

Defn 4. \texttt{var\_struct} structured type for individual variables such that
\[
\texttt{var\_struct} = (\texttt{vkey, type, value})
\]
where
\[
\begin{align*}
\texttt{vkey} &\in \texttt{var\_id\_dom}; \\
\texttt{type} &\in \texttt{class\_id\_dom}; \\
\texttt{value} &\in \texttt{obj\_id\_dom};
\end{align*}
\]
end \texttt{var\_struct}.

Defn 5. \texttt{var\_set} such that \texttt{var\_set = \{var\_struct\}}.

Defn 6. \texttt{method\_id\_dom} identifiers for methods such that \texttt{method\_id\_dom = \{identifier\}}.

Defn 7. \texttt{parameters} type defining formal parameters of a method such that
\[
\texttt{parameters} = \text{sequence of } \texttt{pi} \text{ where } \forall \texttt{pi} : \texttt{t} : \texttt{parameters} = \texttt{pi} \in \texttt{obj\_id\_dom}.
\]
The functions in a method are identified by type and form -- i.e., \texttt{is\_of\_kind} and \texttt{is\_of\_form}. The function \texttt{is\_of\_kind} identifies the functionality or kind of a function. The function \texttt{is\_of\_form}, given the kind of a function, returns the signature or form of the function.

Defn 8. \texttt{method\_struct} structured type which is an identifiable body of functions such that
\[
\texttt{method\_struct} = (\texttt{signature, body})
\]
where
\[
\begin{align*}
\texttt{signature} &\text{ is a (msgkey, param\_list) where } \\
\texttt{param\_list} &\text{ is a parameters; } \\
\texttt{body} &\text{ is a sequence of } f_1, ..., f_n \text{ where } \\
\forall f_i : \texttt{t} &\text{ is \texttt{signature} such that } \\
&\texttt{f1 is\_of\_kind state } \\
&f : \texttt{param\_list\_in x var\_set\_in } \\
&\texttt{var\_set\_out} \Rightarrow \\
&\texttt{f1 is\_of\_form} \\
&f : \texttt{param\_list\_in x var\_set\_in } \\
&\texttt{var\_set\_out} \\
&\texttt{var\_struct} \Rightarrow \\
&f_1 \text{ is\_of\_kind call } \\
&f : \texttt{param\_list\_in x var\_set\_in } \\
&\texttt{var\_set\_out} \Rightarrow \\
&f \text{ is\_of\_form} \\
&\texttt{pass} : \texttt{obj\_id\_dom u class\_id\_dom, obj\_id\_dom u class\_id\_dom, obj\_id\_dom u class\_id\_dom,}
\end{align*}
\]
end \texttt{method\_struct}.

Defn 9. \texttt{class\_pair} is a structured type such that \texttt{class\_pair} = (\texttt{class, superclass}) where
\[
\begin{align*}
\texttt{class} &\in \texttt{class\_id\_dom}; \\
\texttt{superclass} &\in \texttt{class\_id\_dom};
\end{align*}
\]
end \texttt{class\_pair}.

Defn 10. \texttt{msg\_id\_dom} identifiers for messages such that \texttt{msg\_id\_dom = \{identifier\}}.

Defn 11. \texttt{msg\_struct} is a structured type for messages such that
\[
\texttt{msg\_struct} = (\texttt{msgkey, sender, receiver, recipient, contents, control, response_to})
\]
where
\[
\begin{align*}
\texttt{msgkey} &\in \texttt{msg\_id\_dom}; \\
\texttt{sender} &\in \texttt{class\_id\_dom u obj\_id\_dom}; \\
\texttt{receiver} &\in \texttt{class\_id\_dom u obj\_id\_dom}; \\
\texttt{recipient} &\in \texttt{class\_id\_dom u obj\_id\_dom}; \\
\texttt{contents} &\in (\texttt{method\_id\_dom, parameters})^+; \\
\texttt{control} &\in \{\texttt{asynch, synch, response}\}; \\
\texttt{response\_to} &\in \texttt{msg\_id\_dom};
\end{align*}
\]
end \texttt{msg\_struct}.

Defn 12. \texttt{msg\_queue} is a structured type such that
\[
\texttt{msg\_queue} = (\texttt{receiver, mqueue})
\]
where
\[
\begin{align*}
\texttt{receiver} &\in \texttt{obj\_id\_dom u class\_id\_dom}; \\
\texttt{mqueue} &\text{ is a queue of } \texttt{msg\_struct};
\end{align*}
\]
end \texttt{msg\_queue}.

Constraint 1. Identifier partitioning.
\[
\begin{align*}
\texttt{class\_id\_dom n obj\_id\_dom} &\neq \emptyset \land \\
\texttt{class\_id\_dom n method\_id\_dom} &\neq \emptyset \land \\
\texttt{class\_id\_dom n msg\_id\_dom} &\neq \emptyset \land \\
\texttt{class\_id\_dom n var\_id\_dom} &\neq \emptyset \land \\
\texttt{obj\_id\_dom n method\_id\_dom} &\neq \emptyset \land \\
\texttt{obj\_id\_dom n msg\_id\_dom} &\neq \emptyset \land \\
\texttt{obj\_id\_dom n var\_id\_dom} &\neq \emptyset \land \\
\texttt{method\_id\_dom n msg\_id\_dom} &\neq \emptyset \land \\
\texttt{method\_id\_dom n var\_id\_dom} &\neq \emptyset \land \\
\texttt{msg\_id\_dom n var\_id\_dom} &\neq \emptyset.
\end{align*}
\]
Defn 13. Structure of an object, \( o \) such that
\[
\text{obj struct} = (OKey, OState, OPredecessor, OParent, OMeth)
\]
where
\[
\begin{align*}
OKey & \in \text{obj id dom}; \\
OState & \in \text{var set}; \\
OPredecessor & \in \text{obj id dom}; \\
OParent & \in \text{class id dom}; \\
OMeth & \in \text{method id dom};
\end{align*}
\]
end obj struct.

Defn 14. Structure of a class, \( c \) such that
\[
\text{cls struct} = (CKey, Instance_llds, Instance_Var, Class_Var, Method_Set, Superclass_Set, Instances, Versions)
\]
where
\[
\begin{align*}
CKey & \in \text{class id dom}; \\
Instance_llds & \in \text{obj id dom} := \{\}; \\
Instance_Var & \in \text{var set} := \{\}; \\
Class_Var & \in \text{var set} := \{\}; \\
Method_Set & \in \{\text{method struct}\} ; \\
Superclass_Set & \in \text{class id dom} := \{\}; \\
Instances & \text{is a Boolean} ; \\
Versions & \text{is a Boolean} ;
\end{align*}
\]
end cls struct.

Constraint 2. Variable Identifier uniqueness.
Given \( c \) is a cls struct then
\[
\text{Keys(c.InstanceState)} \cap \text{Keys(c.Class_Var)} = \emptyset
\]
The function \( \text{Keys} \), given a set of structures, returns a set of keys of the elements in the structure set.

Constraint 3. Variable value.
\( \neg \) the object whose OKey is the value of a variable must
\( \neg \) be an instance of the class whose CKey is the type of
\( \neg \) the variable
Given \( c_1 \) is a cls struct \( \land c_2 \) is a cls struct then
\[
\forall v: v \in c_1.\text{Instance_Var} \land v \in c_2.\text{Class_Var} : \exists c_2: v.\text{type} = c_2.\text{Okey} : v.\text{value} \in c_2.\text{Instance_llds}.
\]

Constraint 4. Value change.
\( \neg \) type matching for method application
Given \( c_1 \) is a cls struct \( \land c_2 \) is a cls struct then
\[
\forall m : m \in c_1.\text{Method_Set} \land m.\text{body} \text{ is a sequence of } f_1 \ldots f_n; \\
\forall f_i : 1 \leq i \leq m.\text{body} \land f_i.\text{is of kind state} \land f_i.\text{is of form} f : \text{param list in} \times \text{var set in} \to \text{var set out} ; \\
\forall v : v \in \text{var set in} : \exists c_2: v.\text{type} = c_2.\text{Okey} : v.\text{out value} \in c_2.\text{Instance_llds}.
\]

Defn 15. The null value, \( \phi \), indicates that the value for a class variable or instance variable of a class is unknown (\( \phi_1 \)) or is not applicable (\( \phi_2 \)).

Given \( c \) is a cls struct then
\[
\text{ic Method_Set} \geq 1.
\]

Constraint 6. Instantiation.
Given \( c \) is a cls struct then
\[
c.\text{Instances} = \text{false} \Rightarrow c.\text{Instance_llds} = \emptyset.
\]

Constraint 7. Versioning Allowed.
Given \( c \) is a cls struct then
\[
c.\text{Instance} = \text{false} \Rightarrow c.\text{Versions} = \text{false}.
\]

Constraint 8. OPredecessor Value.
Given \( c \) is a cls struct \( \land o \) is a obj struct such that
\( c.\text{OKey} \in c.\text{Instance_llds} \) then
\[
c.\text{Versions} = \text{false} \Rightarrow c.\text{OPredecessor} = \text{null}.
\]

Defn 16. root_class is a cls struct where
\( \text{root_class.\text{Superclass_set}} = \emptyset \).

Defn 17. Structure of a class hierarchy, \( H \) such that
\[
\text{class hierarchy struct} = (O, T, \text{Message_Q, Response_List})
\]
where:
\[
\begin{align*}
O & \in \{\text{obj struct}\} := \{\}; \\
T & \in \{\text{cls struct}\} := \{\text{root_class}\}; \\
\text{Message_Q} & \text{is a msg queue}; \\
\text{Response_List} & \text{is a list of msg struct where} \\
\forall i, j, m: 1 \leq i < 2 \leq \text{Response_List} \land m \text{ Response_List} : m.\text{response_to} < m.\text{response_to} \\
\end{align*}
\]
end class hierarchy struct.

Defn 18. Edges such that
\[
\text{Edges} : \{\text{cls struct}\} \to \{\text{class_pair}\}
\]
where
\[
\text{Edges}(C) = \{c_1, c_2 : C \in C \land c_2.CKey \in c_1.\text{Superclass_set} : \\
\text{CKey} \leq c_1.\text{CKey}, c_2.\text{CKey})\}
\]
end Edges.

Constraint 9. Connectedness.
\( \neg \) all classes have a path to the root_class
Given \( H \) is a class hierarchy structure containing a classbase, \( T \), then
\[
\forall c : c \in T \land c \neq \text{root_class} : \\
\text{is_path}(c.\text{CKey, root_class.\text{CKey}, Edges(T))}.
\]
The function \( \text{is_path} \) determines the existence of a path between two classes in a class hierarchy.

Constraint 10. Acyclicity.
Given \( H \) is a class hierarchy structure containing a classbase, \( T \), then
\[
\forall c : c \in T : \neg \text{is_path}(c.\text{CKey, c.\text{CKey, Edges(T))}.
\]
Constraint 11. Closure.

\(-\) all superclasses must be in system

Given \( H \) is a class_hierarchy_struct containing a classbase, \( C \), then

\[ \forall c \in C : x \in c.\text{Superclass\_Set} : \exists y \in C : x = y.CKey. \]

Constraint 12. Instance variable type.

\(-\) the types of all instance variables are classes within the system (closure).

Given \( H \) is a class_hierarchy_struct containing a classbase, \( C \), then

\[ \forall c_1 \in C : \forall x : x \in c_1.\text{Instance\_Var} : \exists c_2 : c_2 \in C : x.\text{type} = c_2.CKey. \]

Constraint 13. Class variable type.

\(-\) the types of all class variables are classes within the system (closure).

Given \( H \) is a class_hierarchy_struct containing a classbase, \( C \), then

\[ \forall c_1 \in C : \forall x : x \in c_1.\text{Class\_Var} : \exists c_2 : c_2 \in C : x.\text{type} = c_2.CKey. \]

Constraint 14. Unique object key.

Given \( H \) is a class_hierarchy_struct containing an objectbase, \( O \), then

\[ \forall o_1, o_2 : o_1 \in O \land o_2 \in O \land o_1 \neq o_2 : o_1.OKey \neq o_2.OKey. \]

Constraint 15. Unique object parent.

Given \( H \) is a class_hierarchy_struct containing an objectbase, \( O \), and a classbase, \( C \), then

\[ \forall o : o \in O : \exists c : c \in C : o.\text{OKey} \in c.\text{Instance\_Ids} \land o.\text{OParent} = c.CKey. \]

Constraint 16. Instance\_Ids membership.

Given \( H \) is a class_hierarchy_struct containing an objectbase, \( O \), and a classbase, \( C \), then

\[ \forall o : o \in O : \exists k : k \in c.\text{Instance\_Ids} : \exists q : q \in O : q.\text{receiver} = o.\text{OKey} \land q.\text{OParent} = c.CKey. \]

Lemma 1. Each object, \( o \in O \) is an instance of a unique class, \( c \), called a unique parent class.

Proof: By the unique object parent constraint (constraint 15) and the Instance\_Ids membership constraint (constraint 16) which partition the objectbase, \( O \).

Constraint 17. Object method set.

Given \( H \) is a class_hierarchy_struct containing an objectbase, \( O \), then

\[ \forall o : o \in O : o.\text{OMethod} \subseteq \text{Keys}(o.\text{OParent}.\text{Method\_Set}). \]

Constraint 18. Object variable set.

Given \( H \) is a class_hierarchy_struct containing an objectbase, \( O \), then

\[ \forall o : o \in O : o.\text{OState} \subseteq o.\text{OParent}.\text{Instance\_Var}. \]


\(-\) all predecessors of an object come from siblings.

Given \( H \) is a class_hierarchy_struct containing an objectbase, \( O \), then

\[ \forall o : o \in O : o.\text{OPredecessor} \subseteq o.\text{OParent}.\text{Instance\_Ids}. \]

Constraint 20. Message participants.

\(-\) all senders, receivers, and recipients in a message structure must be within the system (closure).

Given \( H \) is a class_hierarchy_struct containing a classbase, \( C \), and an objectbase, \( O \), then

\[ \forall m : m \text{ is a msg_struct} \land m.\text{sender} \in \text{Keys}(H.C) \land m.\text{receiver} \in \text{Keys}(H.C) \land m.\text{recipient} \in \text{Keys}(H.C) \lor \text{Keys}(H.O). \]

Constraint 21. Unique message key.

Given \( H \) is a class_hierarchy_struct then

\[ \forall m_1, m_2 : m_1 \text{ and } m_2 \text{ are msg_struct} \land m_1.mkey \neq m_2.mkey. \]

Constraint 22. Message\_Q contents.

Given \( H \) is a class_hierarchy_struct containing a classbase, \( C \), and an objectbase, \( O \), then

\[ \forall o : o \in H.O \land q \in \text{Message\_Q} : q.\text{receiver} = o.\text{OKey} \land q.\text{CKey} \in H.C \land q \in \text{Message\_Q} : q.\text{receiver} = c.\text{CKey}. \]

Definition 19. Structure of a specification hierarchy, \( S \) such that

\[ \text{spec\_hierarchy\_struct} = (\text{Nodes}, \text{Edges}, \text{Prohibited}, \text{Future}, H) \]

where:

\[ \text{Nodes is a \{cls\_struct\}} = \{\text{root\_class}\}; \]

\[ \text{Edges is a \{class\_pair\}} = \{\}; \]

\[ \text{Prohibited is a \{class\_pair\}} = \{\}; \]

\[ \text{Future is a \{class\_pair\}} = \{\}; \]

\[ H \text{ is a class\_hierarchy\_struct}; \]

end spec\_hierarchy\_struct.

Constraint 23. Nodes subset.

Given \( S \) is a spec\_hierarchy\_struct containing a class hierarchy, \( H \), containing a classbase, \( C \), then

\[ S.H.C \subseteq S.\text{Nodes}. \]

Constraint 24. Unique class key.

Given \( S \) is a spec\_hierarchy\_struct then

\[ \forall c_1, c_2 : c_1 \in S.\text{Nodes} \land c_2 \in S.\text{Nodes} \land c_1 \neq c_2 : c_1.CKey \neq c_2.CKey. \]

Constraint 25. Edges subset.

Given \( S \) is a spec\_hierarchy\_struct containing a class hierarchy, \( H \), containing a classbase, \( C \), then

\[ \text{Edges}(S.H.C) \subseteq S.\text{Edges}. \]
Corollary 1. Given $S$ is a `spec_hierarchy_struct` containing a class hierarchy, $H$, containing a classbase, $T$, then $G_H = \{S.H.G, \text{Edges}(S.H.G)\}$ is a subgraph of $G_S = \{S.Nodes, S.Edges\}$, i.e., $S.H.G \subseteq S.Nodes \land \text{Edges}(S.H.G) \subseteq S.Edges$.

Proof: $S.H.G \subseteq S.Nodes$ by constraint 24. $\text{Edges}(S.H.G) \subseteq S.Edges$ by constraint 25, i.e. $(c_1,c_2) \in \text{Edges}(S.H.G) \Rightarrow (c_1,c_2) \in S.Edges$ where $c_1 \in \text{class_id_dom} \land c_2 \in \text{class_id_dom}$. $\text{root_class.CKey} \in H.G$ is the same as $\text{root_class.CKey} \in S.Nodes$ by definition 16. 

Constraint 26. Faithfulness.
\[ \forall x : x \in S.Edges : x.class \in S.Nodes \land x.superclass \in S.Nodes. \]

Given $S$ is a `spec_hierarchy_struct` then
\[ S.Future \cap S.Prohibited = \emptyset \land S.Future \cap S.Edges = \emptyset \land S.Prohibited \cap S.Edges = \emptyset. \]

Given $S$ is a `spec_hierarchy_struct` then
\[ \forall (c_1,c_2) : (c_1, c_2) \in S.Future : (c_2, c_1) \in S.Future. \]

Defn 20. Function `object-map` maps from an object identifier to the corresponding object such that given an objectbase, $\Theta$, and an objectbase, $\Theta$, then
\[ \text{object-map} : \text{obj_id_dom} \rightarrow \text{obj_struct} \]
where
\[ \text{object-map}(OKey) = o \text{ where } o \in \Theta \land o.OKey = OKey \]
end object-map.

Defn 21. Class behavior is a tuple consisting of the function `object-map`.

Defn 22. Function `instantiate` creates objects such that given $H$ is a `class_hierarchy_struct` containing a classbase, $T$, and an objectbase, $\Theta$, then
\[ \text{instantiate} : \text{class_id_dom} \rightarrow \text{obj_id_dom} \rightarrow \text{such that} \]
\[ \text{instantiate}(K, CKey) \]
where
\[ \exists c : c \in T : c.CKey = CKey \land \]
\[ ((K = \null \land c.Versions = \false) \lor \]
\[ (K = c'.\text{Instance}\_\text{Ids} \land c'.\text{Versions} = \true)) \]
\[ \text{\textbf{-out-assert}} \]
\[ c.\text{Instances} \Rightarrow \]
\[ \text{let } o'\text{new} = (OKey, c'.\text{Instance}\_\text{Var}, \]
\[ c'.\text{Method}\_\text{Set}, c'.\text{CKey}, K) \]
where
\[ o'\text{new}.OKey \neq H.\Theta'\text{in} \land \]
\[ \forall o : o \in H.\Theta'\text{in} : o.OKey \neq o'\text{new}.OKey \land \]
\[ o'\text{new} \in H.\Theta'\text{out} \land \]
o'\text{new}.OKey \in c'.\text{Instance}\_\text{Ids} \land o'\text{new}.OKey \in c'.\text{Out}\_\text{Instance}\_\text{Ids} \land c'.\text{Versions} \Rightarrow \]
\[ \exists o_1 : o_1 \in H.\Theta'\text{in} \land o_1.OKey = K : \forall v_1 : \]
\[ v_1 \in o'\text{new}.\text{OS} \Rightarrow \]
\[ \exists v_2 : v_2 \in o_1.\text{OS} \land v_2.vkey = v_1.vkey : \]
\[ v_1.value = v_2.value \]
end let
end instantiate.

Theorem 1. The objectbase, $\Theta$, of a class hierarchy, $H$, of a specification hierarchy, $S$ is acyclic.

Proof: Given:
\[ O\text{Predecessor} \text{ for an object, } o, \text{ is assigned by the instantiate function at the time the object is instantiated.} \]

The new object, $o$, is assigned an unused object identifier as its $OKey$ -- i.e., the new object's OKey is assigned from $\text{obj_id_dom} \rightarrow \text{Keys}(\Theta)$. 

By contradiction:
Assume some set $O\text{Predecessor}$ which creates a cycle such that $o$ is an element of the set $O\text{Predecessor}$ (which is a subset of some $c.\text{Instance}\_\text{Ids}$ which identifies the OKeys of some subset of objects in the objectbase). Therefore $o$ was already in the objectbase at the time of its creation or instantiation. This contradicts the definition of the instantiate function. Therefore the objectbase is acyclic. 

Corollary 2. There is a partial ordering on all connected $O\text{Predecessor}$ sub-graphs.

Proof: A object, o, of the objectbase draws its OS\text{ate} from the OS\text{ate} of the member(s) of its O\text{Predecessor} set. Each object, o, of the objectbase has its OS\text{ate} defined in terms of objects which preceded it within the Instance\_\text{Ids} set of its parent class c, such that given three objects $o_1, o_2, o_3 \in \Theta$ if
\[ a) \ o_1 \leq o_2 \land \]
\[ b) \text{if } (o_1, o_2) \land (o_2, o_3) \text{ then } (o_1, o_3) \text{ and } \]
\[ c) \text{if } (o_1, o_2) \land (o_2, o_1) \text{ then } o_1=o_2, \]
then there is a partial ordering between $o_1, o_2, o_3$.

Defn 23. Function `response` checks for and retrieves a response from the `Response\_List` such that given $H$ is a `class_hierarchy_struct` containing a `Response\_List` then
\[ \text{response} : \text{msg_id_dom} \rightarrow \text{obj_id_dom} \rightarrow \text{such that} \]
\[ \text{response}(msg) = value \]
where
\[ \exists m : m \in H.\text{Response}\_\text{List} \land m.\text{msgKey} = msg \Rightarrow \]
\[ \text{value} = m.\text{contents} \land \]
\[ \text{value} = \null \] end response.
Definition 24. Function accept accepts and enqueues a message such that given H is a class_hierarchy_struct containing a classbase, C, and an objectbase, O, then

\[
\text{Function accept : msg_struct \rightarrow Boolean}
\]

such that

\[
\text{accept(msg) \equiv B}
\]

where

\[
\begin{align*}
\text{msg.receiver} \in \text{Keys}(H, C) & \Rightarrow \\
B = & \exists c \in H, C \land c = \text{msg.receiver} : \\
\text{msg.method} \in \text{Keys}(c, \text{Method_Set}) & \lor \\
& \text{msg.receiver} \in H, O \\
& \beta = \text{msg.method} \in \text{msg.receiver.OMethod}
\end{align*}
\]

\[
\begin{align*}
& \Rightarrow \\
& \text{out-assert} \\
& \text{enqueue(msg)}
\end{align*}
\]

end accept.

Defn 25. Function apply retrieves a message from the Message_Q and for execution such that

\[
\text{Function apply : obj_id_dom \cup class_id_dom \rightarrow such that}
\]

\[
\text{apply(receiver)}
\]

where

\[
\begin{align*}
\text{not_empty(receiver)} & \Rightarrow \\
& \text{let m = dequeue(receiver)}
\end{align*}
\]

where

\[
\begin{align*}
& \text{apply_aux(m.contents.signature.param_list, m.contents.body)}
\end{align*}
\]

end let

end apply.

Defn 26. Function apply_aux recursively applies the functions of the requested method of a message such that given H is a class_hierarchy_struct containing a classbase, C, and an objectbase, O, then

\[
\text{Function apply_aux : method_struct.signature.param_list, }
\]

\[
\text{method_struct.body \rightarrow such that}
\]

\[
\text{apply_aux(param_list, body) \rightarrow}
\]

where

\[
\begin{align*}
\text{body} & > 1 \Rightarrow \\
& \text{let f = head(body)}
\end{align*}
\]

where

\[
\begin{align*}
& \text{case} \\
& 1 \text{ is of kind state} : \\
& \text{var_set'out = f(param_list'\times \text{var_set'in})}
\end{align*}
\]

\[
\begin{align*}
& 1 \text{ is of kind call} : \\
& \text{answer_list = pass(p1, P2, P3, P4, P5, P6)}
\end{align*}
\]

where

\[
\begin{align*}
p1 & = \text{a Keys}(H, C) \cup \text{Keys}(H, O) \land \\
p2 & = \text{a Keys}(H, C) \cup \text{Keys}(H, O) \land \\
p3 & = \text{a Keys}(H, C) \cup \text{Keys}(H, O) \land \\
p4 & = \text{a (method_id_dom, parameters)} \cup \\
p5 & = \text{a msg_id_dom} \\
p6 & = \text{a msg_id_dom}
\end{align*}
\]

before if p5 = \text{asynch} then

\[
\text{response(answer_list.response_id);}
\]

\[
1 \text{ is of kind value_return} : \\
\text{param_list'out = f(param_list'\times \text{var_set'in})}
\]

end case

end let

before

\[
\text{apply_aux(param_list'out, tail(body))}
\]

end apply_aux.

Defn 27. Function pass creates and passes a message such that given H is a class_hierarchy_struct containing a classbase, C, and an objectbase, O, then

\[
\text{Function pass : Keys(H, C) \cup Keys(H, O), Keys(H, C) \cup Keys(H, O), (method_id_dom, parameters) \cup obj_id_dom, asynch \lor synch \lor response, msg_id_dom \rightarrow Boolean, obj_id_dom, msg_id_dom}
\]

such that

\[
\text{pass(sender, receiver, recipient, contents, control, response_to) \Rightarrow (B, value, response_id)}
\]

where

\[
\text{let msg'new = (msgkey, sender, receiver, recipient, contents, control, response_to)}
\]

where

\[
\begin{align*}
\text{msg.new.control} & = \text{synch} \Rightarrow \\
& (\text{accept(msg'new)} \Rightarrow \\
& \text{response(msgkey) = nil)}
\end{align*}
\]

\[
\begin{align*}
& \text{B = true } \land \text{value = response(msgkey) } \land \\
& \text{response_id = msgkey } \land \text{remove(msgkey)}
\end{align*}
\]

\[
\begin{align*}
& \lor \text{B = false } \land \text{value = null } \land \text{response_id = null } \\
& \lor \text{msg.new.control} = \text{asynch} \Rightarrow \\
& \text{B = accept(msg'new) } \land \text{value = null } \land \\
& \text{response.id = msgkey } \\
& \lor \text{msg.new.control} = \text{response} \Rightarrow \\
& \text{enter(msg'new) } \land \text{B = true } \land \\
& \text{value = null } \land \text{response_id = null}
\end{align*}
\]

subject to

Unique message key constraint.

end let

end pass.

Defn 28. Class hierarchy behavior is a tuple consisting of thirteen functions:

\[
\begin{align*}
& \text{definition 22 instantiate} \\
& \text{housekeeping first} \\
& \text{housekeeping last} \\
& \text{housekeeping not_empty} \\
& \text{housekeeping enqueue} \\
& \text{housekeeping dequeue} \\
& \text{housekeeping enter_answer} \\
& \text{housekeeping remove_answer} \\
& \text{definition 23 response} \\
& \text{definition 24 accept} \\
& \text{definition 25 apply} \\
& \text{definition 26 apply_aux} \\
& \text{definition 27 pass.}
\end{align*}
\]
Defn 29. Function `inherit_properties` causes the inheritance of properties (methods and variables) from a superclass, `c2`, to a subclass, `c1`, such that

\[ \text{Function } \text{inherit_properties} : \text{cls\_struct, cls\_struct } \rightarrow \text{cls\_struct} \]

such that `\text{inherit_properties}(c1, c2) = c1'new` where

\[-I \text{ out-assert} \]
\[ c1'new.\text{Instance}_\text{Var} = c1'old.\text{Instance}_\text{Var} \cup \]
\[ c2.\text{Instance}_\text{Var} \]
\[ c1'new.\text{Class}_\text{Var} = c1'old.\text{Class}_\text{Var} \cup \]
\[ c2.\text{Class}_\text{Var} \]
\[ c1'new.\text{Method}_\text{Set} = c1'old.\text{Method}_\text{Set} \cup \]
\[ c2.\text{Method}_\text{Set} \]
end \text{inherit\_properties}.

Defn 30. Function `class_to_object_inheritance` causes the inheritance of methods and variables from a class, `c`, to its instances such that given `S` is a spec\_hierarchy\_struct containing a class hierarchy, `H`, containing an objectbase, `O`, then

\[ \text{Function } \text{class\_to\_object\_inherit} : \text{cls\_struct } \rightarrow \text{cls\_struct} \]

such that `\text{class\_to\_object\_inherit}(c)` where

\[-I \text{ out-assert} \]
\[ \forall \text{OKey } : \text{OKey } \in c.\text{Instance}\_\text{Ids} : \]
\[ \exists o : o \in S.\text{H}\_\text{O} \land o.\text{OKey } = \text{OKey} \land \]
\[ o.\text{OS}\_\text{new} = o.\text{OS}\_\text{old} \cup o.\text{Instance}_\text{Var} \]
\[ \land o.\text{OM}\_\text{new} = o.\text{OM}\_\text{old} \cup \]
\[ \text{Keys}(c.\text{Method}_\text{Set}) \]
end \text{class\_to\_object\_inherit}.

Defn 31. Function `create_class` creates a class within the specification hierarchy such that given `S` is a spec\_hierarchy\_struct then

\[ \text{Function } \text{create\_class} : \rightarrow \text{cls\_struct} \]

such that `\text{create\_class}() = c'new` where

\[-I \text{ out-assert} \]
\[ c'new \in S.\text{Nodes}'\new \land \]
\[ \forall o : o \in S.\text{Nodes} : c.\text{CKey } \neq c'new.\text{CKey} \land \]
\[ c'new \in S.\text{Nodes}'\\out \]
end \text{create\_class}.

Defn 32. Function `move_class` adds a class to the class hierarchy from the specification hierarchy, such that given `S` is a spec\_hierarchy\_struct containing a class hierarchy, `H`, containing a classbase, `C`, then

\[ \text{Function } \text{move\_class} : \text{cls\_struct, \{class\_pair\}} \rightarrow \text{class\_pair} \]

such that `\text{move\_class}(c1, e)` where

\[-I \text{ out-assert} \]
\[ e.\text{superclass } \in S.\text{Nodes} \land \]
\[ e \in S.\text{Edges} \land \]
\[ e \in S.\text{Future} \land e \in S.\text{Prohibited} \land \]
\[-I \text{ out-assert} \]
\[ \exists c2 : c2 \in S.\text{H}\_\text{in} \land c2 = e.\text{superclass} \land \]
\[ c1.\text{CKey } = e.\text{class} \land \text{Superclass}_\text{Set}'\\new = \{e.\text{superclass}\} \land \]
\[ \text{is\_path}(c1.\text{CKey }, \text{root\_class}\_\text{CKey }, S.\text{Edges}) \land \]
\[-\text{is\_path}(e.\text{class}, e.\text{superclass}, S.\text{H}\_\text{in} \cup \]
\[ \{e\} \land \]
\[-\text{is\_path}(e.\text{superclass}, e.\text{class}, S.\text{H}\_\text{in} \cup \]
\[ \{e\} \land \]
\[ \text{inherit\_properties}(c1, c2) \]
end \text{move\_class}.

Defn 33. Function `create_edge` creates an inheritance relationship between two classes in a specification hierarchy such that given `S` is a spec\_hierarchy\_struct then

\[ \text{Function } \text{create\_edge} : \text{class\_id\_dom, class\_id\_dom } \rightarrow \text{class\_pair} \]

such that `\text{create\_edge}(CKey1, CKey2) = e` where

\[-I \text{ out-assert} \]
\[ \exists C1, C2 : C1 \in S.\text{Nodes} \land \]
\[ C2 \in S.\text{Nodes} : C1.\text{CKey } = CKey1 \land \]
\[ C2.\text{CKey } = CKey2 \land \]
\[-I \text{ out-assert} \]
\[ e.\text{class } = C1.\text{CKey} \land e.\text{superclass } = C2.\text{CKey} \land \]
\[ e \in S.\text{Edges}'\in \land e \in S.\text{Edges}'\out \land \]
\[ e \in S.\text{Prohibited}'\out \land e \in S.\text{Future}'\out \]
end \text{create\_edge}.

Defn 34. Function `move_inheritance_relation` transfers an inheritance relationship from a specification hierarchy to its class hierarchy such that given `S` is a spec\_hierarchy\_struct containing a class hierarchy, `H`, then

\[ \text{Function } \text{move\_inheritance\_relation} : \text{class\_pair } \rightarrow \text{class\_pair} \]

such that `\text{move\_inheritance\_relation}(e)` where

\[-I \text{ out-assert} \]
\[ \exists C1, C2 : C1 \in S.\text{H}\_\text{in} \land C2 \in S.\text{H}\_\text{in} : \]
\[ C1 = e.\text{class } \land C2 = e.\text{superclass} \land \]
\[-\text{is\_path}(e.\text{class}, e.\text{superclass}, S.\text{H}\_\text{in} \cup \]
\[ \{e\} \land \]
\[-\text{is\_path}(e.\text{superclass}, e.\text{class}, S.\text{H}\_\text{in} \cup \]
\[ \{e\} \land \]
end \text{move\_inheritance\_relation}.
Defn 35. Function create_prohibited creates prohibited relationships for a specification hierarchy such that given \( S \) is_a spec_hierarchy_struct then

\[
\text{Function create_prohibited : class_id_dom, class_id_dom} \to \text{class_pair such that } \\
\text{create_prohibited}(\text{CKey}_1, \text{CKey}_2) = e
\]

where

\[
\exists c_1, c_2 : c_1 \in S.\text{Nodes} \land c_2 \in S.\text{Nodes} : \\
c_1.\text{CKey} = \text{CKey}_1 \land c_2.\text{CKey} = \text{CKey}_2 \\
\text{--out-assert} \\
e.\text{class} = \text{CKey}_1 \land e.\text{superclass} = \text{CKey}_2 \land \\
e \in S.\text{Prohibited'in} \land e \in S.\text{Prohibited'out} \\
e \in S.\text{Edges'out} \land e \in S.\text{Future'out} \\
\text{end create_prohibited.}
\]

Defn 36. Function create_future creates an future relationship within a specification hierarchy such that given \( S \) is_a spec_hierarchy_struct then

\[
\text{Function create_future : class_id_dom, class_id_dom} \to \text{class_pair, classpair such that } \\
\text{create_future}(\text{CKey}_1, \text{CKey}_2) = (e_1, e_2)
\]

where

\[
\exists c_1, c_2 : c_1 \in S.\text{Nodes} \land c_2 \in S.\text{Nodes} : \\
c_1.\text{CKey} = \text{CKey}_1 \land c_2.\text{CKey} = \text{CKey}_2 \\
\text{--out-assert} \\
e_1.\text{class} = \text{CKey}_1 \land e_1.\text{superclass} = \text{CKey}_2 \land \\
e_2.\text{class} = \text{CKey}_2 \land e_2.\text{superclass} = \text{CKey}_1 \land \\
e_1 \in S.\text{Future'in} \land e_1 \in S.\text{Future'out} \\
e_2 \in S.\text{Future'in} \land e_2 \in S.\text{Future'out} \\
e_1 \in S.\text{Edges'out} \land e_1 \in S.\text{Prohibited'out} \\
e_2 \in S.\text{Edges'out} \land e_2 \in S.\text{Prohibited'out} \\
\text{end create_future.}
\]

Defn 37. Specification hierarchy behavior is a tuple consisting of eight functions:

- definition 29 inherit_properties,
- definition 30 class_to_object_inherit,
- definition 31 create_class,
- definition 32 move_class,
- definition 33 create_edge,
- definition 34 move_inheritance_relation,
- definition 35 create_prohibited, and
- definition 36 create_future.

Defn 38. A COODB object is an obj_struct such that

\[
\text{COODB object} = \text{obj_struct.}
\]

Defn 39. A COODB class consists of a cls_struct and class behavior such that

\[
\text{COODB class} = (\text{cls_struct, class behavior}).
\]

Defn 40. A COODB class hierarchy consists of a class hierarchy struct where the classbase, \( \mathcal{C} \), and the objectbase, \( \mathcal{O} \), are redefined as

\[
\mathcal{C} \text{ is_a } \{\text{COODB class}\} := \text{(root_class)} \\
\mathcal{O} \text{ is_a } \{\text{COODB object}\} := \{\}
\]

and class hierarchy behavior such that

\[
\text{COODB class hierarchy} = (\text{class_hierarchy_struct, class hierarchy behavior}).
\]

Defn 41. A COODB specification hierarchy is a spec_hierarchy_struct where the class hierarchy subcomponent, \( \mathcal{H} \), is redefined as

\[
\mathcal{H} \text{ is_a } \{\text{COODB class}\} \\
\text{and Nodes is initialized to contain } \\
\text{root_class is_a COODB class}
\]

\[
\text{Nodes} := \{\text{root_class}\}
\]

and specification hierarchy behavior such that

\[
\text{COODB specification hierarchy} = (\text{spec_hierarchy_struct, specification hierarchy behavior}).
\]

Defn 42. A COODB consists of a COODB specification hierarchy, \( S \), such that

\[
\text{COODB} = (S)
\]

where

\[
S \text{ is_a COODB specification hierarchy}.
\]
References


[41] J. Sowa (personal communication).


